Measuring the Maximum Query Throughput of a Single Readyset Node

Readyset Engineering · engineers@readyset.io · measured 14 August 2026 · MySQL protocol, deep cache, fully cached read workload

ABSTRACT

Applications increasingly place a caching layer between the application tier and the database, yet the absolute serving capacity of a production SQL cache is rarely characterized. Published evaluations almost always measure a cache against the database it fronts, which answers a question about the database and terminates as soon as the database saturates. We instead ask how much useful work a single cache node can perform before the cache itself becomes the limiting factor. This isolates the serving path from the database and from cache-fill behavior. While production deployments are constrained by a combination of database throughput, replication, cache misses, and application traffic, understanding the standalone serving capacity establishes the cache's intrinsic ceiling and reveals whether future improvements should target the cache implementation itself or the surrounding system.

A single Readyset server node sustained **4,379,212 queries per second** from cache at **2.26 ms p95** over 2.9 billion requests, with no dropped connections or failed clients. We obtained this by serving a fully cached point-lookup workload over the MySQL wire protocol from 35 independent client machines as load generators, sweeping offered concurrency from 140 to 8,960 concurrent connections — 4 to 256 threads on each generator, one connection per thread. Throughput at least doubled with every doubling of concurrency through 560 connections, and 2,240 connections already delivered 4.1M queries per second, 94% of peak, leaving room to absorb three times as many connections before the hardware saturates.

Cycle-level accounting shows Readyset is not the dominant consumer of CPU at saturation: kernel networking accounts for 56.3% of the machine against 42.8% for all Readyset userspace work, and a symbol-level profile localizes the cache read path itself to approximately 7% of cycles. Because the MySQL protocol is half-duplex, a client cannot issue a second query until it has consumed the first response, and interface counters confirm that nothing batches the resulting round trips away: each query costs one packet inbound and one outbound, to within measurement error, at every concurrency level tested. At peak the node therefore moves 8.7 million packets per second. Per-packet kernel cost is fixed and independent of how fast a lookup is, so the limit follows from protocol structure rather than cache execution, and further gains depend on reducing per-request kernel work or the number of round trips rather than on optimizing lookups.

This result follows a series of optimizations driven by profiling, which eliminated several previously significant userspace bottlenecks in connection scheduling, query execution, caching, tracing, and buffer management. After these optimizations, no single Readyset component dominates execution, and the remaining scalability limit lies outside the cache itself.

This experiment should not be interpreted as a model of a production deployment. Production Readyset nodes simultaneously admit writes, process replication streams, experience cache misses, and serve a mix of query shapes. Those factors determine application-level throughput. Here we intentionally remove them to answer a different engineering question: does Readyset's serving path itself contain an internal scalability bottleneck? The measurements indicate that it does not; the machine reaches its limit in kernel networking before cache execution becomes dominant.

1. Introduction and method

Read-heavy applications routinely interpose a caching layer between the application tier and the system of record, and its value depends directly on how much traffic a single node can absorb. That quantity is poorly characterized, because evaluations comparing a cache against the database it fronts stop measuring the moment the database saturates. We ask instead: *how much work can a single Readyset node perform before Readyset itself becomes the limiting factor?*

Isolating the serving path. Each client issues one primary-key point lookup as a server-side prepared statement on the MySQL binary protocol, prepared once per connection and thereafter only executed. The workload addresses 10,000 keys drawn uniformly from a 1,000,000-row table. The hot set is deliberately small so that memory capacity, eviction and cache-fill behavior are excluded; those are real properties of a cache but belong to a different experiment, and including them would confound the serving cost being isolated.

Why load originates from 35 machines. This is the most consequential methodological decision. A benchmark driven from one host measures that host: a single machine must terminate every connection, take every receive interrupt and run every softirq, so its kernel saturates long before the server's. On this hardware a client whose receive processing is confined to one core saturates near 137,000 queries per second regardless of thread count, and a single-client experiment would report that figure as though it characterized the server. Distributing load across 35 machines places the constraint on the system under test and better resembles production, where reads arrive from many hosts over many connections. It also means the claim can be verified rather than asserted: we sample CPU on all 35 clients throughout, and at peak the server runs at 99.1% while the fleet averages 38.3%, establishing that ~2.5× more load was available than the server could absorb.

The experiment is meaningful only if every request is served from cache, so rather than infer this from throughput we read Readyset's view hit and miss counters immediately before and after each experiment: across the sweep, over 2.9 billion hits and **zero misses**.

Separately, a throughput plateau can be manufactured by mechanisms that resemble saturation without being it — an overflowing accept queue, exhausted backlogs, dropped packets, or a steering imbalance pinning one core. We collect the corresponding kernel counters before and after every experiment; listen overflows, listen drops, backlog drops and time squeezes were all zero throughout.

2. Experimental setup

Role	Host	Configuration
System under test	test-01	AMD EPYC 9454, 48 cores / 96 threads, 251 GiB RAM, Broadcom 10 GbE, Linux 6.8
Upstream database	test-02	MySQL 8.0.46, row-based binlog, FULL row image, 24 GiB buffer pool
Load generators	35 machines	AMD EPYC Genoa, 8 vCPU, 15 GiB RAM, sysbench 1.0.20

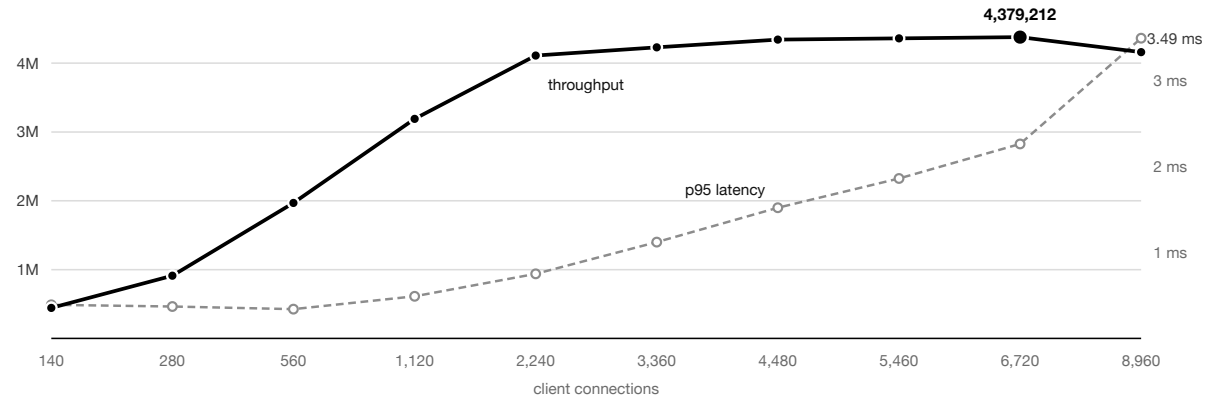
The NIC's combined queues were raised from 24 to the hardware maximum of 74. A receive queue is served by one CPU, so at 24 queues only 24 of the 96 threads handled incoming packets; those CPUs reached about 60% softirq and throughput stopped at roughly 2.97M queries per second while the machine as a whole was 38% idle. Adding queues raised that number immediately, which is how we know the limit was the queue count and not the server. Under load cores sustained 3,705 MHz across all 96 threads at 70 °C against a 95 °C throttle threshold, so results are neither thermally nor frequency limited. Each experiment ran 120 s with 20 s idle between; clients start 0.5 s apart so a connect storm cannot be misread as a server limit. Reported throughput is the sum over clients of each client's median per-interval rate with the first interval discarded, which avoids the staggered ramp depressing the mean.

3. Results

Throughput scales cleanly over the lower half of the range. Between 140 and 560 connections the node delivers 2.06× and 2.16× per doubling of concurrency — marginally superlinear — while per-connection throughput *rises* from 3,164 to 3,514 queries per second and p95 latency *falls* from 0.39 ms to 0.34 ms. The fastest single query recorded anywhere in the fleet was **0.18 ms**, and at 140 connections each connection completes a query every 0.32 ms on average, so most of that time is transit rather than service. Improving latency under increasing load indicates the node is not queueing: at these levels the experiment is bounded by network round-trip time, and added concurrency simply fills otherwise idle time. The node passes one million queries per second at 280 connections, two million at 560, three million at 1,120 and four million at 2,240, with no kernel drop counter advancing at any level.

Figure 1: Throughput and tail latency against offered concurrency

Throughput on the left axis is the primary series; p95 latency on the right axis is secondary. Throughput flattens from 2,240 connections onward while latency continues to climb, so every configuration past that point costs tail latency for throughput the node has already delivered.



Left axis: steady-state throughput. Right axis: median of the per-client p95. Between 2,240 and 6,720 connections throughput gains 6.5% while p95 latency triples, and past 6,720 latency keeps rising as throughput falls.

Connections	QPS	Scaling	QPS / conn	p95	Server CPU	user	sys	softirq	Client CPU
140	442,895	—	3,164	0.39 ms	17.6%	7.3	5.9	4.5	8.1%
280	910,743	2.06×	3,253	0.37 ms	28.4%	11.6	9.0	7.8	16.7%
560	1,967,790	2.16×	3,514	0.34 ms	50.1%	20.2	16.0	13.9	29.1%
1,120	3,189,412	1.62×	2,848	0.49 ms	74.9%	30.1	24.1	20.6	35.2%
2,240	4,111,367	1.29×	1,835	0.75 ms	92.0%	38.1	28.8	25.1	37.2%
3,360	4,229,955	1.03×	1,259	1.12 ms	93.9%	39.5	29.0	25.4	37.2%
4,480	4,343,263	1.03×	969	1.52 ms	95.9%	40.6	29.4	25.9	34.0%
5,460	4,359,420	1.00×	798	1.86 ms	97.2%	41.4	29.8	26.1	38.1%

Connections	QPS	Scaling	QPS / conn	p95	Server CPU	user	sys	softirq	Client CPU
6,720	**4,379,212**	**1.00×**	**652**	**2.26 ms**	**99.1%**	**42.8**	**30.0**	**26.3**	**38.3%**
8,960	4,160,357	0.95×	464	3.49 ms	99.8%	44.7	29.5	25.6	37.3%

Table 1: Complete sweep. Server CPU is decomposed into user, system and softirq; client CPU is the mean across all 35 load generators. Every configuration served a 100% hit ratio, read from Readyset's view hit and miss counters before and after each run.

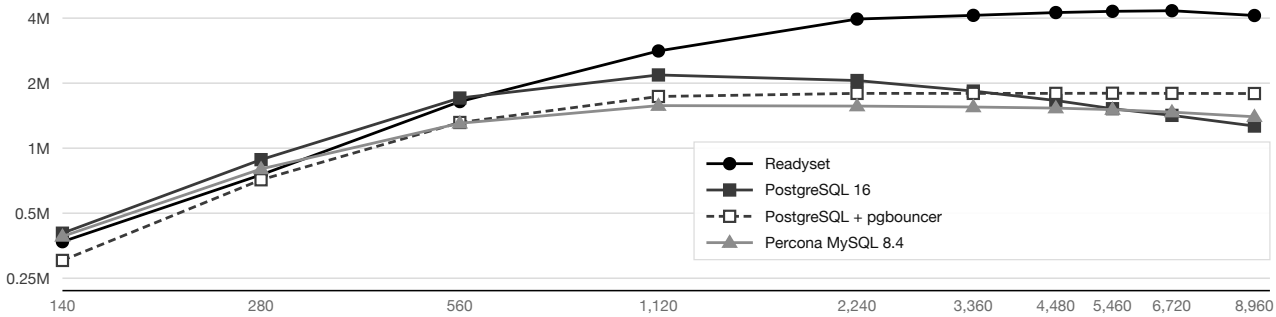
Server CPU rises proportionally with delivered throughput across the entire range, from 17.6% at 442,895 queries per second to 99.1% at 4,379,212. The near-linear relationship is itself a result: it indicates the absence of a lock, queue or contention point whose cost grows superlinearly with load. A system with such a bottleneck shows CPU rising faster than useful work as it approaches saturation; this node does not.

Peak throughput of **4,379,212 queries per second** occurs at 6,720 connections with the machine 99.1% utilized. Beyond it, additional concurrency reduces throughput: at 8,960 connections the node delivers 5% less for 54% more tail latency, the expected signature of scheduling overhead once no idle capacity remains. The efficient operating region sits well below the peak. At 2,240 connections the node delivers 93.9% of peak at 0.75 ms p95 — one third of the tail latency at peak — while retaining 8% CPU headroom. Throughput per unit of Readyset CPU is likewise maximized earlier, at 49,143 queries per second per core at 1,120 connections against 46,243 at peak, a 5.9% difference. Measured from the 2,240-connection configuration, operating at maximum throughput instead trades a three-fold increase in tail latency for a final 6.5% of capacity.

Against the engines a cache fronts. The same workload was run against Percona MySQL 8.4 and PostgreSQL 16 on this hardware, and against PostgreSQL behind pgbouncer, to place the figure above in context. All four configurations were measured in one sweep so that they share host state, fleet and steady-state definition; the Readyset run in that sweep peaked at 4,332,492 queries per second, 1.1% below the figure of Section 3, which came from an earlier sweep of Readyset alone. Readyset peaks at 4,332,492 queries per second, PostgreSQL at 2,181,981 and MySQL at 1,573,565 — factors of 1.99 and 2.75 — at 21.9 µs of machine CPU per query against 42.9 and 60.1. The more consequential difference is shape rather than height. Both traditional engines peak at 1,120 connections and then decline, PostgreSQL losing 42% of its peak by 8,960 connections because its per-query userspace cost rises 70% as the number of connected backends grows; Readyset is still climbing at that point and gives up 5%. Connection pooling addresses the decline rather than the ceiling: pgbouncer costs PostgreSQL 20% of its peak but holds throughput flat to within 0.2% across four doublings of concurrency, because the database continues to see a fixed 1,050 connections however many the clients present.

Figure 2: Serving capacity of four configurations on the same hardware

Log–log axes. Readyset holds its slope furthest before flattening; both PostgreSQL configurations and MySQL turn over at 1,120 connections, and only the pooled configuration stays flat once past its own peak.



Configurations. All four ran on the system under test described in §2, driven by the same 35-generator fleet and the same workload. Readyset: the build discussed in §4, deep cache mode, query logging disabled, client connections spread across 24 runtime shards. PostgreSQL 16: `max_connections` *10,000,* `shared_buffers` *8 GiB, one backend process per connection, JIT and parallel query disabled so a point lookup pays neither. Percona MySQL 8.4:* `thread_handling` *set to pool-of-threads with* `thread_pool_size` *96 and an 8 GiB buffer pool, so connections are multiplexed onto threads rather than mapped one to one. PostgreSQL + pgbouncer: pgbouncer 1.25 in transaction pooling mode with* `max_prepared_statements` *200, six instances on each load generator sharing one port through* `SO_REUSEPORT`*,* `default_pool_size` *5 — so PostgreSQL sees a fixed 1,050 connections however many the fleet offers.*

The poolers are not the constraint in that last configuration. They run on the load generators, which peaked at 48.3% CPU on average and 53.7% on the busiest single machine, while the database host stayed between 98.7% and 99.0% across the whole plateau. A single pgbouncer process is one event loop and would have capped throughput well below what is plotted; six per generator leaves enough headroom that the flat line is PostgreSQL reaching its ceiling at 1,050 connections rather than the pooler reaching its own.

Connections	Readyset	PostgreSQL 16	PostgreSQL + pgbouncer	Percona MySQL 8.4
140	369,198	404,103	302,494	390,035
280	752,315	884,763	714,730	801,171
560	1,642,122	1,707,359	1,315,783	1,303,444
1,120	2,817,532	**2,181,981**	1,736,587	**1,573,565**

Connections	Readyset	PostgreSQL 16	PostgreSQL + pgbouncer	Percona MySQL 8.4
2,240	3,961,475	2,056,529	1,795,027	1,566,044
3,360	4,117,806	1,837,563	1,794,360	1,551,584
4,480	4,241,939	1,663,659	1,794,286	1,533,281
5,460	4,298,904	1,524,197	**1,797,759**	1,508,952
6,720	**4,332,492**	1,417,765	1,794,219	1,468,313
8,960	4,110,854	1,267,028	1,791,781	1,397,903
peak / CPU / cost	4,332,492 99.0% 21.9 µs	2,181,981 97.6% 42.9 µs	1,797,759 98.9% 52.8 µs	1,573,565 98.5% 60.1 µs

Table 3: Queries per second at each concurrency level, with each configuration's peak in bold and its cost per query at that peak. All four were measured on the same host and the same 35-generator fleet within a single day, under the workload described in §1. The Readyset column comes from a later run than Table 1 and on a different client fleet, which is why its peak reads 4,332,492 rather than 4,379,212; the two are within 1.1%. PostgreSQL was measured both directly and behind pgbouncer in transaction pooling mode, with the pooler running on the load generators.

4. Locating the limit

The final measurements reported here are not those of the initial implementation. Throughout this investigation we repeatedly profiled the serving path, identified the largest userspace bottlenecks, and repeated the experiment after removing them. The resulting changes eliminated contention in the adapter, distributed client connections across multiple Tokio runtimes to remove a single-driver bottleneck, reduced hot-path hashing, tracing, logging, and memory-management overheads, and improved request processing on the critical path. The analysis below asks whether, after these optimizations, any internal Readyset bottleneck remains or whether the limiting factor has shifted elsewhere.

A throughput plateau is consistent with several distinct mechanisms: saturation of the load generators, exhaustion of link bandwidth, imbalance in receive-interrupt steering, or exhaustion of server CPU. Each was instrumented independently, and each predicts a different signature in the measurements. The first three are excluded below; the fourth accounts for the observed behaviour.

Not the load generators. At peak the 35 clients average 38.3% CPU while the server is at 99.1%, so the fleet retained roughly 2.5× the capacity consumed and the measurement is bounded by the server. *Not link bandwidth.* The 10 GbE interface moves 3.13 Gbps inbound and 4.87 Gbps outbound, 31% and 49% of line rate; bandwidth is not the constraint, though packet rate is high at 4.35M and 4.36M packets per second. *Not interrupt-steering imbalance.* Per-core utilization at peak spans 97.2% to 99.8%, a spread of 2.6 points across all 96 threads, so no core is saturated while others idle; consistently, softirq flattens near 26% from 2,240 connections onward rather than climbing. The contrasting 24-queue behavior noted in §2 shows what a steering limit looks like on this system, and it does not resemble what we observe.

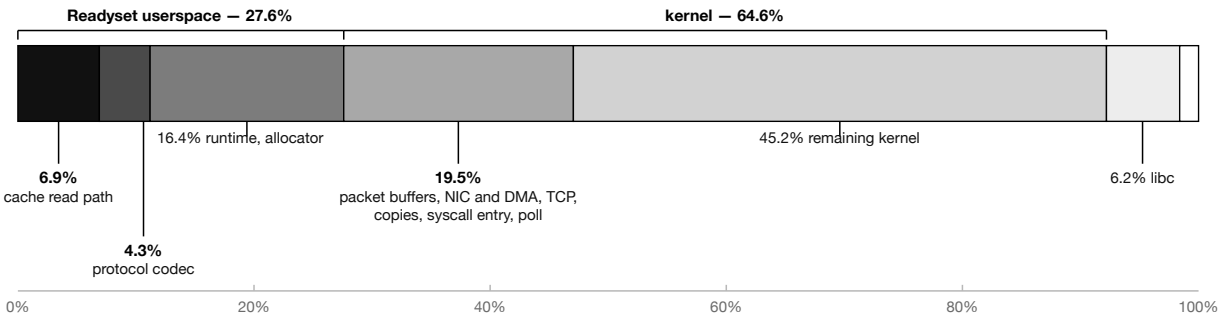
The limit is per-packet CPU cost. The MySQL protocol is half-duplex per connection: a client issues one command and must consume the entire response before issuing another, so there is no pipelining and each query requires its own round trip. Dividing interface packet rates by queries served gives **1.00 packets per query in each direction** — the mean across the ten configurations is 0.996 inbound and 0.996 outbound, with individual levels spanning 0.97 to 1.01, a spread that reflects the error of comparing NIC counters against a client-derived query rate over the same window rather than any real departure from unity. Neither direction benefits from coalescing, so the per-packet cost is paid on every query; requests average 90 bytes and responses 140. At peak the node therefore handles approximately 8.7 million packets per second at an aggregate cost of **21.7 µs of CPU per query** across the 96-thread machine, spending 42.8% of cycles in userspace and 56.3% in the kernel. The majority of the machine is consumed transporting requests and responses rather than answering them.

5. Where the cycles go

Utilization accounting establishes that the kernel dominates but not what Readyset does with its share. We collected a system-wide sampling profile at 999 Hz for 45 s at the 2,240-connection configuration, yielding 4.26 million samples while the node served 4.03 million queries per second at 91.3% utilization and a 100% hit ratio.

Figure 3: Where the machine's cycles go

One hundred per cent of machine cycles at 4.03M queries per second, segmented by what consumes them. Shares are sampled cycles, so they sum to 100 on a single basis.



Sampled-cycle shares apportion userspace against kernel differently from the wall-time decomposition in Table 1 (27.6% / 64.6% against 42.8% / 56.3% at 6,720 connections), because one counts sampled cycles and the other CPU time per core. Both place the majority in the kernel.

Component	Representative symbols	Share of cycles
Cache read path	handle lookup, key hashing, local read, post-processing, view query construction	~6.9%
MySQL protocol codec	packet decode, row writer, flush	~4.3%
Packet buffer allocation	`__slab_free`, `kmem_cache_alloc_node`, `clear_page_erms`	~4.5%
NIC driver and DMA	`bnxt_start_xmit`, `__bnxt_tx_int`, `dma_direct_map_page`	~3.7%
TCP transmit and ack	`__tcp_transmit_skb`, `tcp_clean_rtx_queue`, `tcp_wfree`	~3.2%
User/kernel copies	`_copy_to_iter`, `_copy_from_iter`	~3.2%
Syscall entry and exit	`entry_SYSRETQ_unsafe_stack`	2.75%
Readiness notification	`sock_poll`, `tcp_poll`, `ep_poll_callback`	~2.1%

Table 2: Symbol-level cycle accounting at 4.03M queries per second. Percentages are of all cycles on the machine, not of a single component.

The first row is the result. **The cache read path — locating the reader handle, hashing the key, performing the lookup and post-processing the result — accounts for approximately 6.9% of all cycles.** The work that constitutes cache serving is roughly an eighth of the cost of moving the resulting bytes through the kernel and network interface; adding protocol encoding and decoding brings Readyset's request-specific work to about 11% of the machine.

Two further observations support the conclusion that cache execution is not the constraint. No data structure appears as a scaling bottleneck: the hottest cache-related symbols are hash map operations in the 1.4–1.5% range, with no lock or synchronization symbol in the profile's upper ranks. And readiness notification — a plausible contention point in an event-driven server — costs 2.1%, of which `ep_poll_callback` is 0.70%; in an earlier configuration of this system with fewer runtime shards that single symbol accounted for 5.1% of cycles and was the dominant profile entry. The configuration measured here has moved past that regime with no comparable userspace contention point replacing it.

Instruction-level counters characterize the resulting regime. Over a 10 s window the machine retired 1.61×10^{12} instructions against 3.27×10^{12} cycles, an **IPC of 0.49**, with a 9.47% cache-miss rate. On a core able to retire several instructions per cycle this indicates a memory-latency-bound rather than compute-bound workload, consistent with dominance by per-packet buffer allocation, per-connection socket state across thousands of sockets, and data movement rather than computation over cached data.

6. Discussion

The result of interest is not that a limit exists but where it sits. Across every experiment the cache answered every request from memory, no data structure emerged as a bottleneck, no synchronization point appeared in the profile, and per-core utilization stayed uniform to within 2.6 points at saturation. The system stopped scaling because the machine ran out of CPU while spending most of it on packet transport.

This has a specific consequence for how such a system should be improved. Optimizing cache execution addresses a component measured at roughly 7% of cycles; eliminating it entirely could not produce a large change in throughput. The dominant costs — syscall entry and exit, packet buffer allocation, driver and DMA work, TCP processing, user/kernel copies — are incurred per packet, and the protocol fixes the packet count at one per query in each direction. Improvements must therefore come from reducing per-request kernel work or from changing the number of round trips, not from faster lookups. The protocol's half-duplex structure binds the latter: batching is unavailable to a client performing single point lookups, and reducing bytes rather than packets offers little, since at 31% and 49% of line rate compression would trade CPU, the scarce resource, for bandwidth, the abundant one. Interfaces that amortize kernel entry across many operations, such as `io_uring`, target the syscall component directly; our profile bounds it at roughly 4% of cycles including the readiness path, which calibrates the expected benefit.

Finally, throughput and latency are asymmetric here in a way that matters operationally. Throughput is flat across the upper half of the concurrency range while tail latency grows monotonically, so the configuration that maximizes throughput is not the one most deployments should choose: 2,240 connections retains 93.9% of peak at 0.75 ms p95 and preserves headroom for replication and load variation, whereas the peak configuration triples tail latency and leaves 0.9% idle.

7. Limitations

- **Single query shape and small hot set.** A primary-key lookup whose projection is the constant `1`, over 10,000 keys of a 1,000,000-row table. Larger results would shift the packet-to-query ratio, and the small hot set says nothing about capacity, eviction or cache-fill behavior.
- **Read-only.** No concurrent write load; behavior under concurrent replication is not characterized.
- **Single measurement per configuration.** Repeated experiments at 280 connections differed by 6.8%, so low-concurrency, round-trip-bound points show meaningful variance; saturated points are substantially more stable.
- **One platform, with the AMD SRSO mitigation and the host firewall disabled** for the measurement window on a single-tenant test machine. These are not deployment recommendations. The 10 GbE interface was not the limit here, but at this packet profile egress would approach line rate near 9 million queries per second.

8. What is Next

The experiment isolates one code path: a single query shape, read-only, on the binary protocol with a statement prepared once per connection and executed thereafter. Nothing else competed for the machine, but it also means the result characterises one path rather than the system. The same method extends to the paths it excluded, and each answers a question this one cannot. We plan to experiment with (but not limited to) the below list:

Writes in the path. Nothing wrote to the dataset here. A thread updating rows continuously would exercise the incremental update path that keeps a cache up-to-date, and would answer two separate questions: how many updates per second the node absorbs before update processing becomes the constraint, and how much read throughput a given update rate costs. The second is the figure a deployment actually needs. Reads and writes contend for the same cores and the same reader state, and the size of that interference is currently unmeasured.

More complex query shapes. The point lookup was chosen to minimise everything except serving cost. A wider shape, joins, aggregation, a larger result set, would move work into parts of the dataflow this profile barely touches, and would change the packet-to-query ratio that fixes the present ceiling. Because statements are prepared once per connection, we expect parse and plan cost to stay off the hot path, so any difference should come from result construction and from the bytes moved rather than from query analysis.

The text protocol. Every figure here used server-side prepared statements on the binary protocol, where a query costs a bound parameter and no SQL text. Clients that send text instead pay parsing and transformation on every execution, and this profile gives no basis for estimating what that costs at these rates. We expect a substantial reduction. Quantifying it matters because a large share of real traffic arrives that way, and a ceiling that only holds for prepared statements is a narrower claim than it appears.

Each of these adds a variable to an experiment whose value came from having almost none, so each should be run against the same fleet, the same counters and the same steady-state definition used here, otherwise the comparison against this baseline is lost.

9. Conclusion

A single Readyset node sustains over 4.3 million fully cached SQL queries per second at 2.26 ms p95, and stops there because the machine runs out of CPU rather than because the cache does. At that point kernel networking accounts for 56.3% of the host against 42.8% for all Readyset userspace work, and the cache read path proper for approximately 7% of cycles.

The practical consequence is that further throughput depends on reducing per-request kernel work or the number of round trips, not on faster lookups: a component measured at 7% of cycles cannot be optimized into a materially higher ceiling.

Sysbench 1.0.20, server-side prepared statements on the MySQL binary protocol · 35 × 8-vCPU load generators, 120 s per configuration, 20 s idle between, clients staggered 0.5 s apart · hit ratio from Readyset view hit and miss counters sampled per experiment · CPU from `mpstat`, network from `sar`, kernel drop counters from `nstat` and `softnet_stat`, profile from `perf` at 999 Hz · steady-state throughput is the sum over clients of per-client median interval rates with the first interval discarded · all figures recomputed from archived raw captures · measured 14 August 2026.